Automated Verification of Functional Interface Compatibility

Stefan Kolatzki, Ursula Goltz, Matthias Hagner, Andreas Rausch, Björn Schindler

29.05.2012

NTH Computer Science Report 2012/01

NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

niedersächsische technische hochschule

Technische Universität Braunschweig

TU Clausthal
Clausthal University of Technology

Leibniz Universität Hannover

IMPRESSUM

Publisher

Niedersächsische Technische Hochschule (NTH)
Technische Universität Clausthal, Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany

Editors of the Series

Prof. Dr. Christian Müller-Schloer,
Prof. Dr. Andreas Rausch,
Prof. Dr. Lars Wolf

Technical Editor

Dr. Sebastian Herold
Contact: sebastian.herold@tu-clausthal.de

NTH Computer Science Report Review Board

Prof. Dr. Jiří Adámek
Prof. Dr. Jürgen Dix
Prof. Dr. Ursula Goltz
Prof. Dr. Jörg Hähner
Dr. Michaela Huhn
Prof. Dr. Jörg P. Müller
Prof. Dr. Christian Müller-Schloer
Prof. Dr. Wolfgang Nejdl
Dr. Dirk Niebuhr
Prof. Dr. Niels Pinkwart
Prof. Dr. Andreas Rausch
Prof. Dr. Kurt Schneider
Prof. Dr. Christian Siemers
Prof. Dr. Heribert Vollmer
Prof. Dr. Mark Vollrath
Prof. Dr.-Ing. Bernardo Wagner
Prof. Dr. Klaus-Peter Wiedmann
Prof. Dr.-Ing. Lars Wolf

ISBN 978-3-942216-17-3

Content

ABSTRACT

A well established approach for managing the complexity of large-scale software systems is to use a component based approach to achieve structure and modularity. During the lifetime of such systems, necessary changes are typically carried out by replacing components by other components, for adaptive systems even at runtime. For this it is crucial to make sure that new components are compatible with the rest of the system. This means not only that interfaces should be compatible, but beyond that to ensure functional compatibility. In this paper, we suggest to use a combination of testing and model checking to check functional compatibility. Therefore, the formal model DisCComp and its description technique is used to specify the semantic requirements of component interfaces explicitly and in particular in an operational style. The Java PathFinder model checker is used to verify the validity of a system consisting of certain chosen components automatically. This approach enables efficient coverage techniques to verify component compatibility even at runtime.

1 INTRODUCTION

Component-based software engineering (CBSE) [18][21]has been continuously improved and successfully applied over the past years changing the predominant development paradigm: Systems are no longer developed from scratch, but composed of existing, reusable software 'parts' called software components. Thus CBSE promises to enable practical reuse of software components as well as more flexible, adaptable and evolvable software systems by component replacement. We use Clemens Szyperski's definition of a component, which is widely accepted: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” ([18], page 34).

This definition has several implications. The component needs a clear description of the component including its provided and required interfaces (see Fig. 1). This description has to be delivered together with the component by the component vendor [18]. Such a description is crucial for a component to be composable with other components by a third party, the component user.

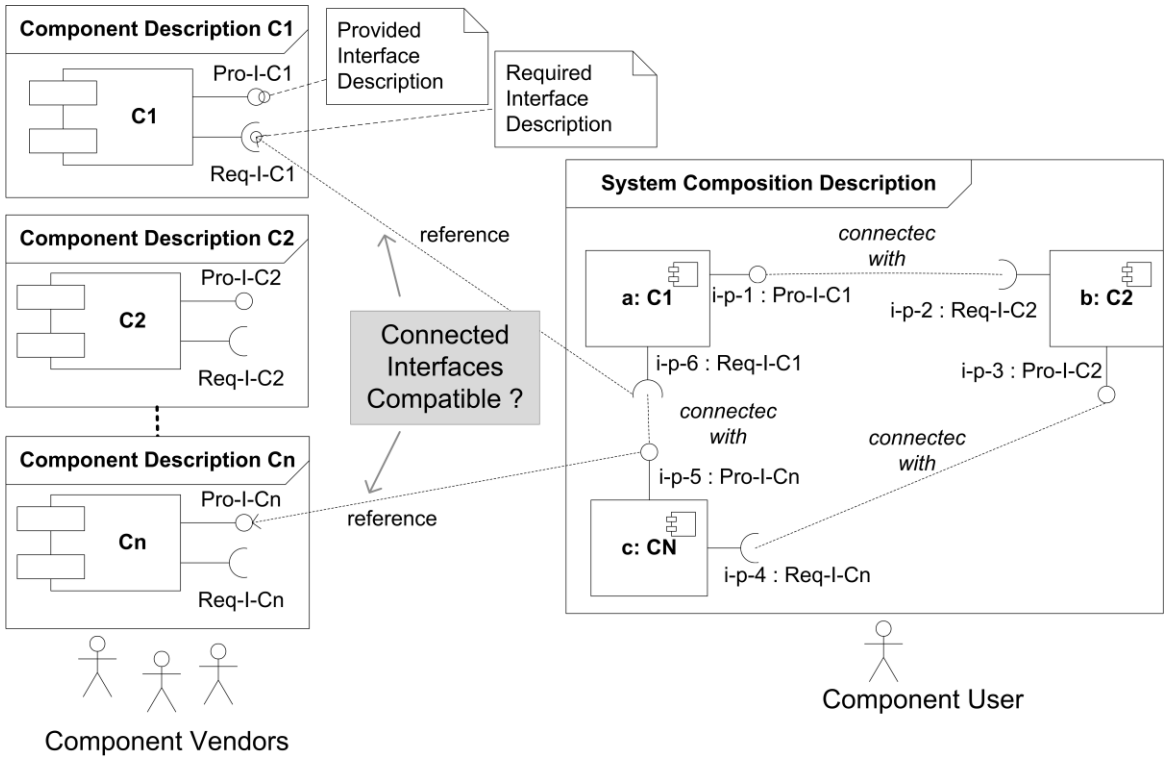

FIG. 1. COMPONENTS AND SYSTEM COMPOSITION

Whenever a component-based system is composed together from two or more components, the component user defines the system composition description [5] as shown in Fig. 1. The system composition description contains a connection of required and provided interfaces of the components under composition. To verify the correctness of component composition means to check whether connected interfaces are compatible. Therefore, various concepts and methods have already been developed and successfully applied, like f.i. Interaction Contracts [7], Reuse Contracts [17], Evolving Interoperation Graphs [12], and Requirements/Assurances Contracts [13].

However, all these methods and techniques share a common basic approach: To verify the correctness of the system composition one has to prove, that the connected required and required interfaces are compatible. At this, properties described in required interfaces have to be guaranteed by the provided properties in the connected provided interfaces. Most of the description techniques used in the above

mentioned approaches for the description of interface properties have a high level of expressiveness. They use f.i. first order logic, temporal logic, or any other Turing complete description technique. Consequently, verifying the correctness of the system composition results in a proof of equivalence or at least a refinement relation between two turing machines. It is well known that this is undecidable and cannot be automatized [19], [4].

Hence, in this paper we suggest to use the formal model DisCComp and its description technique, introduced in [14], to specify the semantic requirements of required component interfaces explicitly and in particular in an operational style. This allows us to use a combination of testing and model checking approach to check functional interface compatibility. Our main goal is to automatically verify the functional correctness of a system composition, even at runtime. Even though, this verification cannot be complete (due to the undecidable issue), nevertheless it should go as far as possible.

This paper is structured as follows. Section 2 gives an overview of the initial situation of our approach. In section 3 we give a short introduction to DisCComp and section 4 introduces a running example for the paper. Our main approach is described and discussed in section 5. Section 6 concludes the paper.

2 OVERALL APPROACH: AUTOMATED VERIFICATION OF FUNCTIONAL INTERFACE COMPATIBILITY

Recent trends and developments in research and industry, like for instance ultralarge scale systems share a common future trend [6]: Complex software systems are no longer considered to have well defined boundaries. Instead future software systems consist of a vast array of distributed, autonomous, heterogeneous, cooperating and continually evolving subsystems resp. components. Components may join or leave these systems during their whole life cycle even during runtime. We call those systems dynamic adaptive component-based systems. Therefore a common component infrastructure, like for instance MoCA [16], MundoCore [2], and DAiSI [9], is required that supports the system composition and component binding at runtime.

To achieve a certain degree of composition correctness in dynamic adaptive component-based systems with respect to hot plug and composition of components during runtime we have elaborated a runtime testing approach (cf. [10] and [11]). This approach has been integrated into our component infrastructure DAiSI (Patent pending. Patent Nr. 10 2008 050 843.8, 8.10.2008). Following this, the component vendor has to describe for each required interface a set of test cases. These test cases are executed before component composition to test their compatibility. This technique has been successfully used in research and industry demonstrators, like f.i. an Emergency Management System [15].

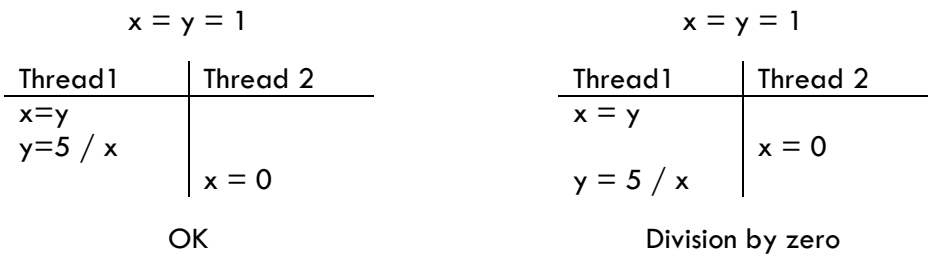

FIG. 2. SCHEDULING LEADS TO DIFFERENT BEHAVIOUR

However, this first step towards a more sophisticated runtime component composition verification approach has some drawbacks. The scope of the verification is limited to the scope of the test cases. Additionally, every test case considers exactly one test path with a single interleaving. As shown in Fig. 2 this might lead to unfound errors, caused by unconsidered interleavings. If the scheduler decides to execute the statements in thread 2 before those in thread 1, the value of the variable X is zero, causing a division by zero. If only the first scheduling is checked by the testing environment, this error is not found resulting in a false positive.

For that reasons we propose within this paper an approach to apply automated verification techniques, like dynamic model checking [20] [8], to verify the correctness of component composition. Dynamic model checking enables higher test path coverage with consideration of various interleavings.

Usual components interface descriptions are a non operational and Turing complete, like f.i. pre- and post-condition based interface descriptions (see left hand side of Fig. 3). A proof of compatibility between a description of a required interface and a description of a provide interface is not decidable as discussed in Section 1. To apply a testing based approach, like dynamic model checking, we have to execute the description. Consequently, we use an executable operational description for the provided interfaces as shown on the right hand side of Fig. 3. The descriptions of the required interfaces contain the properties to be verified by the model checker. Based on this interface description approach dynamic model checking can be used to verify component composition correctness.

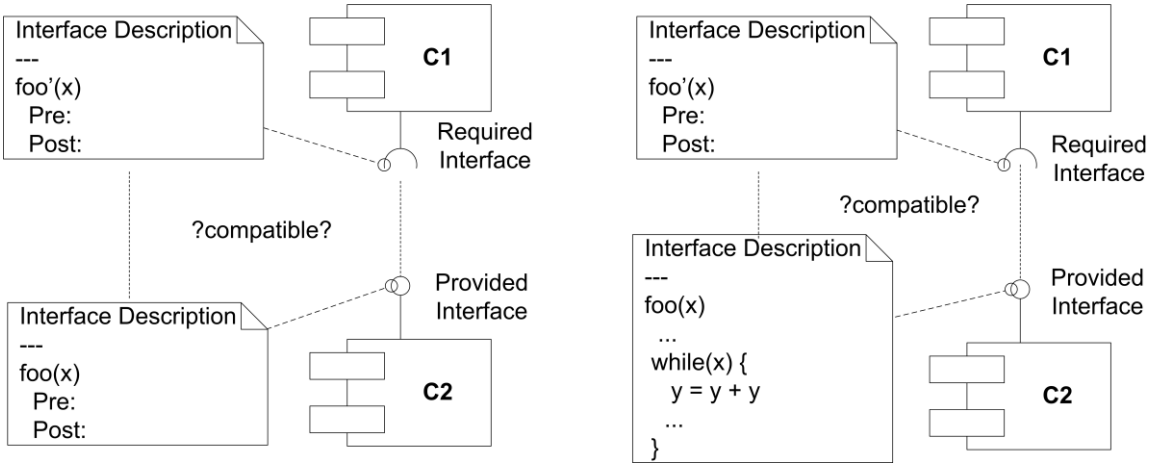

FIG. 3. OPERATIONAL INTERFACE DESCRIPTION

3 DISCCOMP COMPONENT MODEL

DisCComp is a formal model based on set-theoretic formalizations of distributed concurrent components. It allows modelling of dynamically changing structures, a shared global state and asynchronous message communication, as well as synchronous method calls. DisCComp provides a sound semantic model for concurrently executed components that is realistic enough to serve as a formal foundation for component technologies currently in use (e.g. CORBA, J2EE). We provide a UML-based description technique for structural and behavioural aspects of component-based systems. With the DisCComp approach, a software system comprises a set of disjoint instances at runtime: system, component, interface, attribute, connection, message, thread, and value. A component can be connected to other components by interfaces, which encapsulate the behaviour of the components. Using this connection network, asynchronous messages can be send and it is also possible to access attributes of interfaces.

A system may change its structure dynamically: instances may be created or deleted, attributes may be assigned to interfaces, interfaces may be assigned to components, or connections between interfaces can be created or deleted. The description of a state of a system consists of the structure of the system, a state of the communication, and the values of the attributes. This state is denoted as a snapshot.

The DisCComp approach focuses on execution streams instead of timed streams. Whenever a thread's call stack changes (e.g. a method return or a new method call) a new observation point is reached. As it is done in techniques like CORBA or J2EE, there is a global order of all method calls and return. Consequently, there is an order of all observation points and at every observation point, a snapshot is created, capturing the state of the system. A transition between two states of the system is between a certain part of the system-wide snapshot and a certain part of the threads' wished system-wide successor snapshot after performing a method call or return.

Thus, we need some specialised runtime system that schedules all threads at each new method call or return from a method call. Whenever a thread wants to perform a new method call or return, which means that its behaviour relation fires, the run-time system composes a new well-defined system-wide successor snapshot based on the thread's requested changes and the current system-wide snapshot.

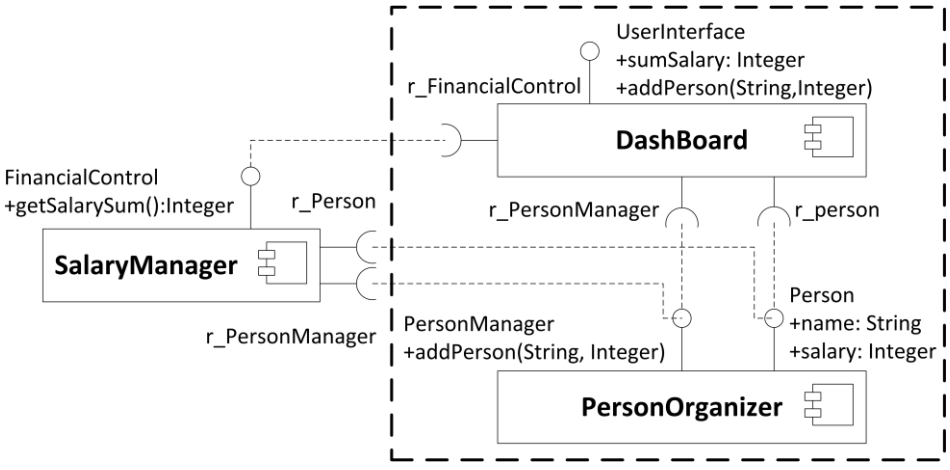

FIG. 4. SIMPLE HUMAN RESOURCE MANAGEMENT SYSTEM

Fig. 4 gives an excerpt from a Human Resource Management System, containing components for handling persons, salaries, and an external entry point. Three components are represented: **DashBoard**, **SalaryManager**, and **PersonOrganizer**. For simplicity, we consider only the two components **Dashboard** and **PersonOrganizer** (marked by the dotted line). They share two interfaces ("PersonManager" with the method *addPerson()* and "Person" with the possibility to access the stored persons). One requirement is,

that an object of type “Person” that is passed as parameter of the function *addPerson()* is really added to the system with the correct name and salary.

4 DISCCOMP DESCRIPTION TECHNIQUE EXAMPLE

An automated testing based approach for the example introduced in Section 3 requires a component description. In this section the example is described by the DisCComp Description Technique. At this, the description is represented by a concrete textual syntax.

A representation of the description by a graphical language based on the Unified Modeling Language (UML) [1] is also feasible. The code In Fig. 5, Fig. 6, and Fig. 7 is the textual illustration of the example in Fig. 4.

```
COMPONENT DashBoard
PROVIDED
   INTERFACE UserInterface [1,1]
      CONNECTION personManagerOfDB END thePersonManager: r_PersonManager [1,1]
      METHOD addPerson(name: String, salary: Integer)
         thePersonManager.r_addPerson(name, salary);
REQUIRED
   INTERFACE r_PersonManager [1,1]
      CONNECTION r_personsOfPM END r_persons: Person [0,*]
      MESSAGE r_addPerson(r_newName: String, r_newSalary: Integer)
           POST correctPersonStorage()
              self@PRE.r_persons.size() + 1 == self.r_persons.size();
              IF(self.r_persons.exists(p: r_Person | NOT
                    self@PRE.r_persons.exists(p2: r_Person | p == p2)) )
              THEN (p.r_name == r_newName) AND (p.r_salary == r_newSalary);
              ELSE false; ENDIF;
   INTERFACE r_Person [0,*]
      ATTRIBUTE r_name: String
      ATTRIBUTE r_salary: Integer
```

FIG. 5. COMPONENT DASHBOARD

The component description of **DashBoard** in Fig. 5 defines the following properties: **DashBoard** has a provided interface "UserInterface" and the required interfaces "r_PersonManager" and "r_Person". **UserInterface** has a connection *thePersonManager* to the interface "r_PersonManager" and a method *addPerson()*. "r_PersonManager" requires a set of connections to "r_Person" interfaces. Additionally, it requires a message handling of *r_addPerson()*. "r_Person" requires the attributes *r_name* and *r_salary*.

At method invocation *addPerson()* of "UserInterface" a message *r_addPerson()* is sent to *thePersonManager*. The description of "r_PersonManager" assigns the following post condition to the message processing of *r_addPerson()*: One "r_Person" has to be added to the set of "r_Person" connections after *r_addPerson()* is processed. Furthermore, *r_name* and *r_salary* of the added "r_Person" has to be equal to *r_newName* and *r_newSalary* received by the message *r_addPerson()*.

```
COMPONENT PersonOrganizer
PROVIDED
   INTERFACE PersonManager [1,1]
      CONNECTION personsOfPM END persons: Person [0,*]
      MESSAGE addPerson(name: String, salary: Integer)
         newPerson: Person := NEW Person;
         newPerson.name := name;
         newPerson.salary := salary;
         newPersonsOfPM: personsOfPM := NEW personsOfPM BETWEEN newPerson AND self;
   INTERFACE Person [0,*]
      ATTRIBUTE name: String
      ATTRIBUTE salary: Integer
```

FIG. 6. COMPONENT PERSONMANAGER

The component description of **PersonOrganizer** in Fig. 6 defines the following properties: **PersonOrganizer** has the provided interfaces "PersonManager" and "Person". "PersonManager" has a set persons of connections to "Person" interfaces and can receive the message *addPerson()*. "Person" has the attributes *name* and *salary*. At reception of the message *addPerson()* a new interface "Person" is created with the name and salary received by the message. The new "Person" interface is added to the set of connections *persons*.

```
SYSTEM SystemToVerify
   USED COMPONENTS DashBoard, PersonOrganizer
   INITIALIZATION
      theDB: DashBoard := NEW DashBoard;
      thePO: PersonOrganizer := NEW PersonOrganizer;
      theUI: UserInterface:= NEW UserInterface ASSIGNED TO theDB;
      thePM: PersonManager:= NEW PersonManager ASSIGNED TO thePO;
      connUsPm: personManagerOfDB:=NEW personManagerOfDB BETWEEN theUI AND thePM;
      theUI.addPerson("Richard Paulson",1200);
   MAPPING OF DashBoard::r_PersonManager TO PersonOrganizer::PersonManager
      MAP r_personsOfPM TO personsOfPM; MAP r_addPerson TO addPerson;
   MAPPING OF DashBoard::r_Person TO PersonOrganizer::Person
      MAP r_name TO name; MAP r_salary TO salary;
```

FIG. 7. SYSTEM DESCRIPTION

The description of the system, depicted in Fig. 7, contains an initialization of the components and the interfaces as defined in Fig. 4. At this, the interfaces are assigned to components. Furthermore, the system description contains a mapping from the required to the provided interfaces.

5 PATHFINDER BASED EVALUATION

The basic idea of our approach is to execute the whole description using the Java PathFinder (JPF) [3]. JPF is an explicit-state model checker for Java implemented programs. It can examine all possible executions of a program due to nondeterministic choices and different thread interleavings. JPF implements a backtracking Java Virtual Machine that executes Java byte code using partial order reduction to reduce the state explosion.

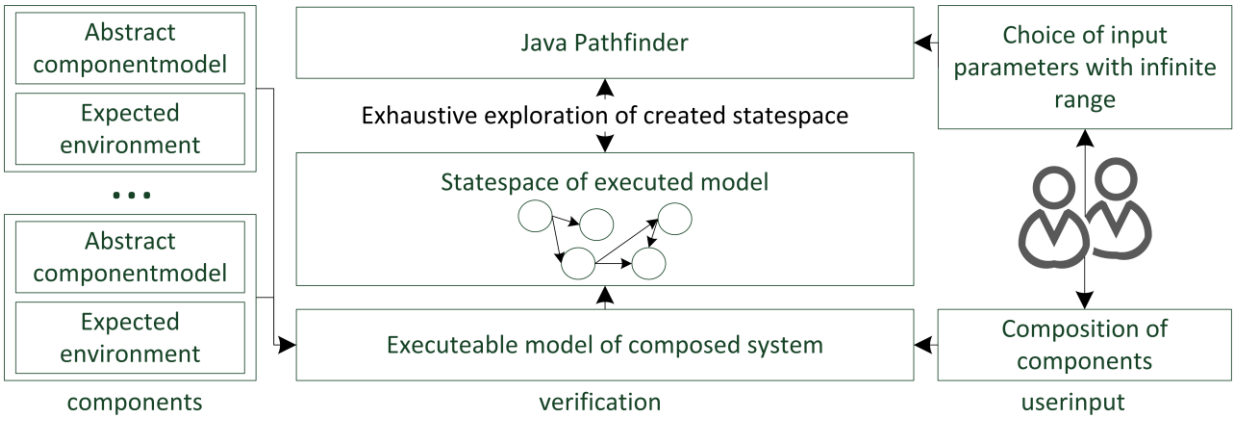

FIG. 8. EXPLORATION OF THE CREATED STATESPACE

As JPF checks all possible execution states, it will be able to find errors like deadlocks or uncaught exceptions. Check methods, containing the pre, post conditions and the invariants, are executed by a DisCComp runtime environment every time a snapshot is created. These checks throw `ViolationExceptions` if a required condition is not met which are caught by JPF, recognized as an error, and then being displayed to the user. For this approach, the whole system is necessary, even if only two interfaces are checked, as there could be side effects or requirements depending on the provided parts of other components, not under consideration.

Fig. 8 gives an overview of our approach. On the left side it shows the components, originally specified with DisCComp and transformed into Java code. In the middle, the verification process with JPF is sketched. On the right side, the user input is depicted which is necessary to limit the possible state space and to build up the composition of the components.

Additionally, Fig. 8 depicts the limitations of this approach: the coverage of the verification depends on the user given input parameters and, depending on the complexity and the user input, the state space could be infinite and the verification process will not terminate. The input parameters have to be chosen with respect to the infinite state space explosion, but covering all relevant cases. Future work will consider complexity and decidability in detail.

If an input parameter has a type with a finite domain, e.g. an integer, JPF is able to check it exhaustively, by selecting all values consecutively. In case of an infinite input parameter, e.g. a string without size limitation, the user has to limit the domain by manually selecting a finite number of values out of it.

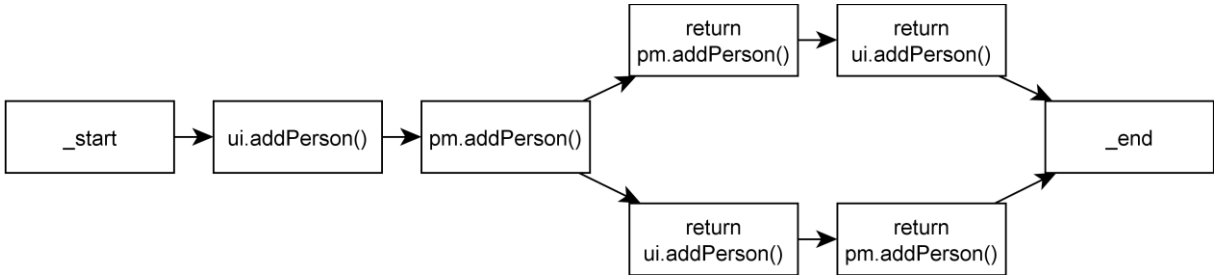

FIG. 9. STATESPACE EXAMINED WITH JPF

JPF creates a state space depending on the snapshots of the described system. Every snapshot, created when a method starts, returns, or sends a message, leads to a new state created by JPF, as described in the definition of DisCComp. In Fig. 9 the possible states for the running example are given. In this case, there are two different schedules possible. The first schedules executes return *pm.addPerson()* before executing return *ui.addPerson()* as the second schedule does this the other way around. The other calls are executed sequentially, because the callees are defined as synchronous methods.

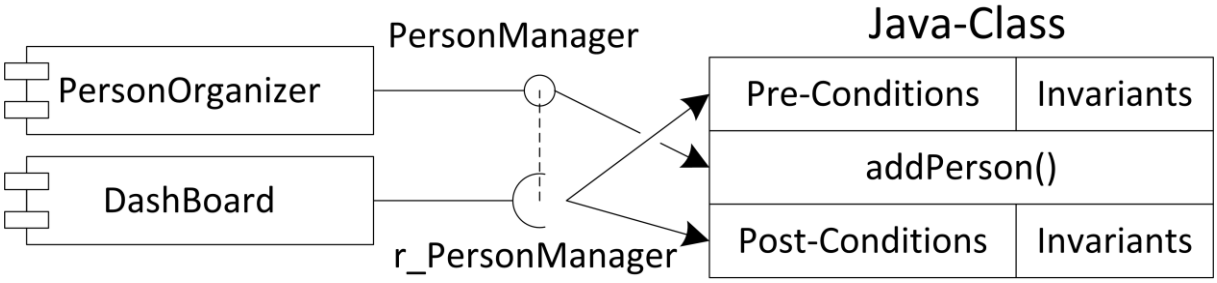

FIG. 10. INTERWEAVING OF THE CONDITION CHECKS AROUND A METHOD

Every interface is represented as a Java-Class, including the provided methods. The methods contain the provided operational description from the component description. Additionally, there are check methods, containing the pre- and post-conditions and invariants, which are taken from the required interface description of the other components depending on the provided interface. For example, depicted in Fig. 10, **PersonOrganizer** offers the Interface "PersonManager" with the Method *addPerson()*. Consequently, in our approach, there is a Java-Class *PersonManager* containing a method *addPerson()* with the operational implementation from the provided part of the interface description.

```
====================================================== coverage statistics
bytecode line branch methods location
-------------------------------------------------------------------------------
1,00(45/45) 1,00(12/12) - 1,00 (2/2) personorganizer.PersonManager
1,00(41/41) 1,00(10/10) - - addPerson( String,Integer )
-------------------------------------------------------------------------------
1,00(45/45) 1,00(12/12) - 1,00 (2/2)
====================================================== results
no errors detected
====================================================== search finished
```

FIG. 11. COVERAGE OF JPF RUN

On top of that, there is a method that checks if the required conditions, described in the component description of Component **DashBoard**, are met (e.g. if the person is added to the system and has the correct name and salary). At each snapshot, the runtime environment performs a check, whether the method behaved according to the assumptions of the other components, using the corresponding check methods and the current state.

```
public void addPerson(String name, Integer salary) {
   CreateInterfaceStep createPerson = new CreateInterfaceStep("Person",
       getParentComponent());
   createPerson.proceed();
   Interface newPerson = createPerson.getCreatedInterface();
   SetAttributeStep setName = new SetAttributeStep(newPerson, "name", name);
   setName.proceed();
   SetAttributeStep setSalary = new SetAttributeStep(newPerson, "salary", salary);
   setSalary.proceed();
   CreateConnectionStep createConn = new CreateConnectionStep("personsOfPM", this,
       newPerson);
   createConn.proceed();
}
```

FIG. 12. METHOD ADDPERSON() TRANSLATED FOR THE VERIFICATION

The code in Fig. 12, showing the *addPerson()* method of the interface "PersonManager", gives an idea how the DisCComp description looks like in our verification environment. The Java code is the result of a manual translation of the DisCComp specification. We plan to do this automatically in the future.

JPF offers a statistical feedback visualizing the coverage of our example and showing it complete. Fig. 11 shows the statistics of a successful verification run. The coverage in this example is limited to the method *addPerson()* in the component "PersonManager", but JPF offers the ability to collect coverage information from all interfaces under consideration.

```
====================================================== system under test
application: SystemInit.java
====================================================== search started
...
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
runtime.exceptions.PostConditionViolatedException: MethodReturnStep (41586)
   returning from PersonManager.addPerson([Richard Paulson, 1200]) to
   PersonManager in Thread[Thread-0,5,main] causes:
correctPersonStorage(): Information of person was not stored correctly.
Environment:
[0]  COMPONENT DashBoard
INTERFACE UserInterface (43540)
        CONNECTION PersonManagerOfDB to PersonManager 43219
...
====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "PostCondition..."
====================================================== search finished
```

FIG. 13. CONDITIONVIOLATION REPORTED BY JPF

During execution, JPF recognizes errors as uncaught exceptions, thrown by the runtime environment. Fig. 13 shows the output of the verification process with a provoked error, by not setting the attribute salary in the method *addperson()*. JPF catches the thrown `PostConditionViolatedException` and cancels the verification process. The implementation of this special exception gives information about which condition was not met and the sequence of created snapshots to the user.

6 CONCLUSION AND FUTURE WORK

In this paper, we used the formal model DisCComp and its description technique to specify the semantic requirements of component interfaces in an operational style. We introduced a combination of testing and model checking to verify functional compatibility of components, using the Java PathFinder and a self-developed runtime environment coupling it to the DisCComp description technique. We illustrated the feasibility of our approach by a rather simple case study. Future work will consider complexity and decidability in detail. For more extended applications, the performance of our framework needs to be improved, for example by more elaborated scheduling techniques.

At the current state of development, the whole system is always required, when trying to verify compatibility. This effort might be reduced by trying to first check two components against each other. If the verification fails, e.g. because of side effects with other components, the next component is added to the system and then checked again. The result might be a verification of a partial system, which could be reused in following verification instances.

Another direction for further research is to automatically generate test suites, which now still have to be provided by the vendor of a component, for example by means of symbolic execution.

REFERENCES

1. OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.2 (2009), http://www.omg.org/spec/UML/2.2/Superstructure
2. Aitenbichler, E., Kangasharju, J., Mühlhäuser, M.: MundoCore: A Light-weight Infrastructure for Pervasive Computing. Pervasive and Mobile Computing 3(4), 332-361 (2007)
3. Brat, G., Havelund, K., Park, S., Visser, W.: Java pathfinder - second generation of a java model checker. In: Proc. of Workshop on Advances in Verification (2000)
4. Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics 58, 345-363 (1936)
5. Crnkovic, I., Chaudron, M., Larsson, S.: Component-based development process and component lifecycle. Int. Conf. on Software Engineering Advances, 44 (2006)
6. Feiler, P et al., D., Sullivan, K., et al.: Ultra-large-scale systems - the software challenge of the future. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (2006)
7. Helm, R., Holland, I., Gangopadhyay, D.: Contracts: Specifying behavioral compositions in object-oriented systems. SIGPLAN Not. 25, 169-180 (1990)
8. King, J.: Symbolic execution and program testing. Commun. ACM 19, 385-394 (1976)
9. Niebuhr, D., Klus, H., Anastasopoulos, M., Koch, J., Weiß, O., Rausch, A.: DAiSI- Dynamic Adaptive System Infrastructure. Tech. rep., Fraunhofer Institut für Experimentelles Software Engineering (2007)
10. Niebuhr, D., Rausch, A.: Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems based on Runtime Testing. In: Proc. of the 4th Workshop on Services Integration in Pervasive Environments (SIPE 09) at the Int. Conf. On Pervasive Services 2009 (2009)
11. Niebuhr, D., Rausch, A., Klein, C., Reichmann, J., Schmid, R.: Achieving dependable component bindings in dynamic adaptive systems - a runtime testing approach. IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 186-197 (2009)
12. Rajlich, V.: Modeling software evolution by evolving interoperation graphs. Ann. Softw. Eng. 9, 235-248 (2000)
13. Rausch, A.: Software evolution in componentware using requirements/assurances contracts. In: Proc. of the 22nd Int. Conf. on Software engineering (2000)
14. Rausch, A.: Disccomp _ a formal model for distributed concurrent components. Electron. Notes Theor. Comput. Sci. 176, 5-23 (2007)
15. Rausch, A., Niebuhr, D., Schindler, M., Herrling, D.: Emergency Management System. In: Proc. of the Int. Conf. on Pervasive Services 2009 (2009)
16. Sacramento, V., Endler, M., Rubinsztejn, H.K., Lima, L.S., Goncalves, K., Nascimento, F.N., Buenos, G.A.: MoCA: A middleware for developing collaborative applications for mobile users. IEEE Distributed Systems Online 5, 2 (2004)
17. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse contracts: managing the evolution of reusable assets. In: Proc. of the 11th ACM SIGPLAN Conf. on Object- Oriented Programming, Systems, Languages, and Applications (1996)
18. Szyperski, C.: Component Software, Beyond Object-Oriented Programming. Addison Wesley Publishing Company (2002)
19. Turing, M.: On computable numbers, with an application to the entscheidungsproblem. In: Proc. of the London Mathematical Society (1936)
20. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proc. Of the 15th IEEE international conference on Automated software engineering (2000)
21. Kozaczynski, W., Booch, G., “Component-Based Software Engineering”, IEEE Software valume: 155, Sept.-Oct. 1998, pp. 34-36